Ranking Cartesian Sums and K-maximum subarrays

Sung Eun Bae and Tadao Takaoka
seb43,tad@cosc.canterbury.ac.nz
Department of Computer Science and Softwoare Engineering
University of Canterbury, New Zealand

November 19, 2006

Abstract

We design a simple algorithm that ranks K largest in Cartesian sums $X + Y$ in $O(m + K \log K)$ time. Based on this, K-maximum subarrays can be computed in $O(n + K \log K)$ time (1D) and $O(n^3 + K \log K)$ time (2D) for input array of size n and $n \times n$ respectively.

1 Introduction

Selecting the K-th largest in a list of size n is a well-known problem and a linear time worst-case solution is known [1]. Another approach is to build a binary heap and repeatedly output/delete the root K times. This approach takes $O(n+K \log n)$ time, less efficient than [1], but we have K elements in sorted order by default. If we wish to get the same result in sorted order based on [1], we first find the K-th largest, discard elements smaller than the K-th and sort the remaining. It results in $O(n + K \log K)$ time. When the order is required, it can be shown that both approaches are equivalent, but the heap-based one is substancially simpler.

We consider the selection of the K-th element in a set of Cartesian sums $X + Y = \{x_i + y_j | x_i \in X, y_j \in Y\}$, where $X = \{x_1, x_2, .., x_n\}$, $Y = \{y_1, y_2, .., y_m\}$. Frederickson and Johnson [2] gave an optimal solution with $O(m + p \log(K/p))$ time, where $n \leq m$ and $p = \min\{K, m\}$. When $K \leq m$, this is linear time. We look at the original problem in a different angle, and wish to find the K largest elements in $X + Y$ in sorted order. Based

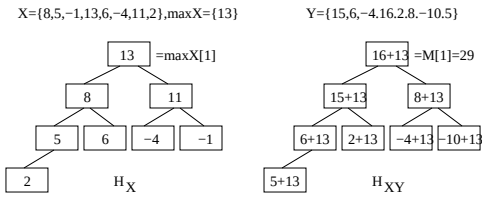

Figure 1: Build max-heaps H_X and H_{XY} to solve $X + Y$ problem

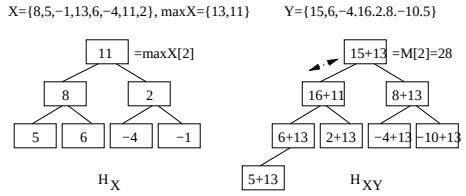

Figure 2: Get the next maximum in $X + Y$

on [2], it inevitably introduces extra $O(K \log K)$ term in the complexity due to sorting. We design a simple algorithm for this problem using the heap-based approach, which is asymptotically equivalent to [2] when the order is considered. This is easily extended to a general selection in $X_1 + X_2 + .. + X_d$. We also apply this reult to design an algorithm for K-maximum subarray problem, both one-dimensional (1D) and two-dimensional (2D) versions.

2 $X + Y$ Problem

Let $maxX[w]$ be the w-th maximum in X. Let XY be a list of size m such that, $XY = \{y_1 + maxX[1], y_2 + maxX[1], .., y_m + maxX[1]\}$. The first largest in $X+Y$ is then of course, $M[1] = \text{MAX}\{XY\}$. Note that we use MAX , MIN for operator to avoid confusion with list names containing lowercase max or min. We build a max-heap H_X with X and obtain $\text{MAX}\{X\}$ at the root, which we store in $maxX[1]$. The rest in $maxX$ are yet unknown. Then we build another max-heap H_{XY}, where each node contains $y_j + maxX[1]$ $(j = 1..n)$. Here, the root of H_{XY} is $M[1]$ (Figure. 1). Let $M[1]$ be $y_j + maxX[1]$ for some j. We

update the root of H_{XY} with the next maximum obtainable with y_j, which is $y_j + maxX[2]$. In general, when the root containing $y_j + maxX[w]$ is updated, we check if $maxX[w+1]$ is readily available in $maxX$. Otherwise, the current root of H_X is $maxX[w]$, thus we delete the root of H_X and take the new root as $maxX[w+1]$. Now we update H_{XY} and obtain $M[2]$ from the root (Figure 2). We repeat this process K times and output K largest sums in $X + Y$ in sorted order. The correctness of the algorithm is easily observed.

Building two heaps take linear time, and each subsequent maximum is found in logarithmic time. Hence, the total time is $O(m + K \log m)$. Note that the maximum value for K is mn in the extreme, but if $K \leq n \leq m$, we can reduce the size of X and Y to K by leaving only the K largest elements in X and Y. This can be done by the linear time selection algorithm [1] without increasing the complexity. The total time then becomes $O(m + K \log K)$, which is asymptotically equivalent to [2] plus sorting. We conclude the total time is $O(n + K \log \min(K, m))$.

This algorithm can be easily generalised to the selection in $X_1 + X_2 + .. + X_d$. We can prepare d heaps in a similar manner. If K is not greater than the size of any X_i $(1 \leq i \leq d)$, we achieve $O(n + Kd \log K)$ time, where n is $|X_1| + |X_2| + .. + |X_d|$.

3 K-maximum subarray problem

For a given array $a[1..n]$ containing a mixture of positive and negative real numbers, the maximum subarray is the consecutive array elements of the greatest sum. Let $Max(K, L)$ be the operation that selects the K largest elements in a list L in non-increasing order. When $L = \{a[i] + a[i+1] + .. + a[j] | \forall\ i, j\ where\ 1 \leq i \leq j \leq n\}$, we wish to obtain $M = Max(K, L)$, and the location of each $M[k]$ $(k = 1..K)$.

Example a ={3, 51, -41, -57, 52, 59, -11, 93, -55, -71, 21, 21}. Among them, $M[1]$ is 193, $a[5] + .. + a[8]$ if the first element is indexed 1. We denote this by $193(5, 8)$. $M[2] = 149(1, 8)$ and $M[3] = 146(2, 8)$ etc.

This problem was first discussed in [3] and an optimal solution of $O(n+K\log\min(K,n))$ time is given in [4, 5]. We design another algorithm with the same complexity. Note that we study *overlapping* K maximum subarrays. Strictly disjoint maximum subarrays are discussed in [6, 7].

We start with the prefix sum sum's of a given input $a[1..n]$, such that $sum[0]=0$ and $sum[i]=a[1]+a[2]+..+a[i]$. The sum of arbitrary consecutive elements, $a[i]+a[i+1]+..+a[j]$ is easily computed by $sum[j]-sum[i-1]$. We define $min_i[w]$ as the w-th minimum of $\{sum[0],..,sum[i-1]\}$. Let a list $Cand$ be $\{sum[1]-min_1[1], sum[2]-min_2[1],..,sum[n]-min_n[1]\}$.

The first maximum sum $M[1]$ is then $\mathrm{MAX}\{Cand\}$. Suppose $M[1]$ is $sum[i]-min_i[1]$ for some i. We update this i-th entry in $Cand$ by replacing $min_i[1]$ with $min_i[2]$. $M[2]$ is then the new maximum of $Cand$. Similar to the $X+Y$ problem, we can build H_{Cand}, a max-heap with elements in $Cand$ to facilitate the maximum selection.

However, the maintenance of min_1,..,min_n is not trivial. In $y_j+maxX[w]$, assuming that $maxX[w]=x_i$ for some i, we are not concerned of the position of x_i in X. However, in $sum[i]-min_i[w]$, assuming $min_i[w]=sum[v]$ for some v, the position of v must be in the range of $0\le v\le i-1$. Also, $maxX[w]$ is the list-wide w-th largest element in X. But $min_i[w]$ is the w-th minimum in the sub-list specific to $sum[i]$, i.e. $\{sum[0],..,sum[i-1]\}$. In the example above, one can easily observe that $min_4[1](=0)\neq min_5[1](=-44)$.

To overcome the difficulty, the easiest option is to create an individual min-heap for each min_i $(i=1..n)$. This is, however, too costly. Still, if we use a *persistent tournament* to maintain multiple versions of min_i, we can show that the followings hold.

Lemma 3.1. *All $min_i[1]$'s $(i=1..n)$ can be computed in $O(n)$ time.*

Lemma 3.2. *A tournament for min_i can be prepared in $O(\log n)$ time.*

Lemma 3.3. *The next element in min_i can be obtained in $O(\log n)$ time.*

We first present Algorithm 1 assuming that all the lemmas hold.

Algorithm 1 Computing K maximum subarrays. Results are in $M[1..K]$ in sorted order.

1: **for** $i \leftarrow 1$ to n **do** Compute $min_i[1]$, the minimum of $sum[0], .., sum[i-1]$
2: Build min-tournament T_0 with $sum[0]$, $sum[1]$,..,$sum[n-1]$.
3: Build max-heap H_{Cand} with $sum[i] - min_i[1]$ for all $i = 1..n$.
4: **for** $k \leftarrow 1$ to K **do**
5: $M[k] \leftarrow root(H_{Cand})$. Output $M[k]$. Suppose $M[k]$ is $sum[i] - min_i[w]$.
6: $min_i[w+1] \leftarrow$ the next unconsumed minimum in $\{sum[0], .., sum[i-1]\}$
7: Replace $root(H_{Cand})$ with $sum[i] - min_i[w+1]$ and update H_{Cand}
8: **end for**

Lemma 3.1 is trivial. Line 1 runs a sequential scan on $sum[0], .., sum[i-1]$ and computes "prefix minimum" for each position i. Initially, we only know $min_i[1]$'s for all $i = 1..n$, but $min_i[w]$ $(w > 1)$ will be found when it is needed. Now we describe the techniques to support Lemma 3.2 and 3.3.

3.1 Creating a persistent tournament

A tournament is similar to a *heap* in terms of its feature and computational complexity. When n items are present, there are $O(n)$ nodes in the structure, and the maximum (or minimum) is located at the root. It takes $O(n)$ time to build, and basic operations take $O(\log n)$ time. An advantage of a tournament over a heap is the preservation of the comparison history and the locational information. We can examine the tree to learn who beats who, and the original position of the final winner. Keeping such information in a heap may be difficult, if not impossible.

We build a min-tournament with $sum[0], .., sum[n-1]$. Each node contains the smaller value of two children. The overall minimum is placed at the root. Let us refer to this tournament as T_0. Each node also maintains the *coverage*, derived from the range of indices of prefix sums under its control. A node covering $sum[i]$,..,$sum[j]$ has a coverage $(i+1, j+1)$.

To maintain min_i $(i = 1..n)$, we need the i-th version T_i that covers $sum[0], .., sum[i-1]$. However, we wish to avoid building each version of tournament from scratch. We borrow the *node copying* technique used in *persistent* data structure, which allows access to the old

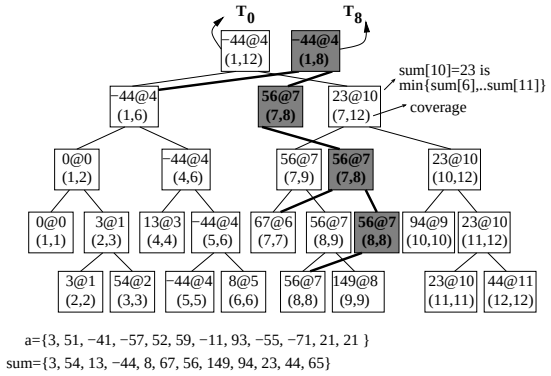

Figure 3: Retrieving T_i with T_0 kept intact. T_0 is built with the input in Example. T_i is the tournament that maintains min_i. Here, the root of T_8 is $min_8[1] = -44$, that is $sum[4]$

versions after subsequent update operations [8]. We show how to retrieve T_i from T_0, and update T_i while keeping T_0 and other versions intact.

3.1.1 Preparing the i-th version

To maintain min_i, we prepare T_i, the i-th version of the tournament. We want the root of T_i to have a coverage $(1, i)$. We visit the i-th leaf (containing $sum[i-1]$) in T_0 and traverse back towards the root. If the currently visiting node has a right sibling, we copy the parent node and let this parent not have a right child. The coverage of the copied parent will not go above i. When we arrive at the root, the copied root has the coverage $(1, i)$ with the value MIN $\{sum[0], .., sum[i-1]\}$, i.e. $\min_i[1]$. Now T_i is ready for use. During the process, as we only updated the copied nodes, T_0 is kept intact. Note that most nodes in T_i are *original*, created when T_0 was built. Only those copied nodes are *version specific* to T_i. The retrieval of T_i from T_0 took $O(\log n)$ time, proving Lemma 3.2.

3.1.2 Updating the i-th version

Suppose we have discovered $min_i[1..w]$, and now wish to find $min_i[w+1]$. If $w = 1$, T_i is not available yet, so we retrieve it from T_0 as described above. Otherwise, the current root of T_i is $min_i[w]$. Let $min_i[w] = sum[x]$ for $x < i$. We access the root of T_i and traverse

from the root of T_i to the $(x+1)$-th leaf that contains $sum[x]$. We replace this leaf with ∞, and update the rest nodes on the path to the root. We are allowed to update a node version specific to T_i. Otherwise, we make a copy and update it. This routine again is $O(\log n)$ time, proving Lemma 3.3.

3.2 Analysis of Algorithm 1

Lines 1,2 and 3 are linear time. Building a tournament is recursively done. Line 5 is $O(1)$ time. Line 6 involves at most two $O(\log n)$ time operations. If T_i is already available, we simply access this. Otherwise, we retrieve it from T_0 spending $O(\log n)$ time. Line 7 involves the max-heap maintaining $Cand$, and takes $O(\log n)$ time to return each of $M[k]$ and update the heap. Altogether, the total time is $O(n+K \log n)$. Note that K can be $\frac{n(n+1)}{2}$ in the extreme, and this algorithm can work any K. Bengtsson and Chen [5] observed that $O(n+K \log n) = O(n+K \log K)$, hence the total time is $O(n+K \log \min(K, n))$, matching the previously known best results [4, 5]. The algorithmic structure of our algorithm is similar to [5], but it is easier to extend to higher dimensions with less complexity.

4 K-maximum subarrays in two dimensions

For an array of size $n \times n$, we search for a rectangle that contains the maximum sum. For a single maximum sum, we can extend the algorithm for 1D and make an $O(n^3)$ time solution. Slightly sub-cubic time algorithms are also known [9, 10]. For K maximum sums, [11] showed that $O(n^3)$ time or even sub-cubic time is sufficient with some constraints on K. Most notably, Cheng et. al [4] achieved $O(n^3 + K \log n)$ time algorithm, which is $O(n^3)$for $K \leq \frac{n^3}{\log n}$.

For an array a of size $m \times n$ $(m \leq n)$, We first compute the prefix sum $sum[1..m][1..n]$, where $sum[i][j]$ is sum of $a[1..i][1..j]$. This is easily computed in $O(mn)$ time, and used throughout the process.

A *strip* is a subarray of a with horizontally full length n, covering multiple rows. For

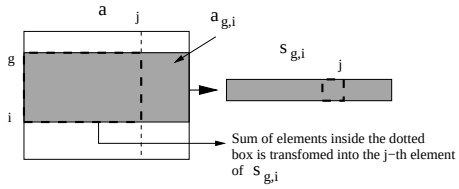

Figure 4: Separating a strip $s_{g,i}$ from the 2D input array

example, a strip $a_{g,i}$ is $a[g..i][1..n]$ that covers row $g..i$. We denote variables associated with this strip with the subscript, such as $M_{g,i}$. A strip can be transformed into a 1D prefix sum array, which can be processed by an algorithm for 1D.

We compute the prefix sum of a strip $a_{g,i}$, which is denoted by $sum_{g,i}$ (Figure 4). For any j, $sum_{g,i}[j]$ can be computed by $sum[i][j] - sum[g-1][j]$. For all pairs of g, i ($g \leq i$), we obtain $sum_{g,i}[1..n]$'s in $O(m^2 n)$ time.

Each $sum_{g,i}$ is processed by lines 1-3 of Algorithm 1. As a result, there are $O(m^2)$ max-heaps, $H_{Cand_{g,i}}$. We collect all $M_{g,i}[1]$'s, the root of each $H_{Cand_{g,i}}$ into a list $2Cand$ and build another max-heap H_{2Cand}. The first maximum sum in 2D, $M[1]$, is located at the root of H_{2Cand}. Suppose $M[1]$ is $M_{g,i}[1]$. To get $M[2]$, we obtain $M_{g,i}[2]$ after updating $H_{Cand_{g,i}}$. Then we replace the root of H_{2Cand} with $M_{g,i}[2]$ and update this heap, whose new root returns $M[2]$. This is repeated K times.

Each update operation to H_{2Cand} is $O(\log m)$ time. Finding the next maximum sum involves an update to one $H_{Cand_{g,i}}$ followed by an update to H_{2Cand}. As $m \leq n$ is assumed, each subsequent maximum sum is found in $O(\log \min(K, n))$ time. The total time is $O(m^2 n + K \log \min(K, n))$, which is cubic time if $m = n$ and $K \leq n^3 / \log n$, matching the previous result [4]. In general, in a d-dimensional array of size $n \times \cdots \times n$, there are $O(n^{2d-2})$ 1D problems. An extra heap H_{dCand} maintains the maximum sum of each 1D problem, and we achieve $O(n^{2d-1} + K \log \min(K, n))$ time. While $K = O(n^{2d})$ in extreme, it is simply $O(n^{2d-1})$ time for $K \leq \frac{n^{2d-1}}{log \min(K,n)}$.

5 Concluding remark

We gave a simple heap-based algorithm for ranking K maxima in $X+Y$, and extended this idea to the K-maximum subarray problem. They match the previously known results and present a useful framework for similar problems involving multiple lists.

References

[1] M. Blum, R. Floyd, V. Pratt, R. Rivest, R. Tarjan, Time bounds for selection. J. Computer and System Sciences 7(4) (1973) 448-461.

[2] G. Frederickson, D. Johnson, The complexity of selection and ranking in X+Y and matrices with sorted rows and columns, J. Computer and System Sciences 24 (1982) 197-208

[3] S. E. Bae, T. Takaoka, Algorithms for the problem of K maximum sums and a VLSI algorithm for the K maximum subarrays problem, in: Proc. of ISPAN 2004, 2004, pp.247–253.

[4] C. Cheng, K. Chen, W. Tien, K. Chao, Improved algorithms for the k maximum-sums problems, in: Proc. of ISAAC 2005, LNCS 3827, 2005, pp.799–808.

[5] F. Bengtsson, J. Chen, A note on ranking k maximum sums. Tech. Rep. 2005:08 Luleå University of Technology

[6] W. L. Ruzzo, M. Tompa, A linear time algorithm for finding all maximal scoring subsequences, in: Proc. of ISMB'99, 1999, pp. 234-241.

[7] S. E. Bae, T. Takaoka, Algorithm for k disjoint maximum subarrays, in: Proc. of the International Conference on Computational Science (ICCS 2006), 2006, Part I, pp. 595-602

[8] J. R. Driscoll, N. Sarnak, D. D. Sleator, R. E. Tarjan, Making data structures persistent, in: Proc. of STOC'86, 1986, pp. 109-121

[9] H. Tamaki, T. Tokuyama, Algorithms for the maximum subarray problem based on matrix multiplication, in: Proc. of SODA 1998, pp.446-452.

[10] T. Takaoka, Efficient algorithms for the maximum subarray problem by distance matrix multiplication. Elec. Notes in Theoretical Comp. Sci., Vol.61 Elsevier, 2002.

[11] S. E. Bae, T. Takaoka, Improved algorithms for the K-maximum subarray problem. Computer Journal 49 (3) (2006) 358–374.